AD-A110 861 STANFORD UNIV CA COMPUTER SYSTEMS LAB F/G 9/2
DIRECTLY EXECUTED LANGUAGES.(U)
JAN 82 M J FLYNN DAAG29-78-G-0205
UNCLASSIFIED ARO-15877.2-EL NL

END
DATE FILMED
8-82
DTIC

LEVEL II

DIRECTLY EXECUTED LANGUAGES

Final Report

Prepared for

Department of the Army
U.S. Army Research Office
Research Triangle Park, NC

Contract No. DAAG29-78-G-0205

September 25, 1978 - September 24, 1981

by

Michael J. Flynn
Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford,CA 94305

The views and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A110 861	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DIRECTLY EXECUTED LANGUAGES		5. TYPE OF REPORT & PERIOD COVERED Final Report 25 Sept. 1978 - 24 Sept.1981
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael J. Flynn		8. CONTRACT OR GRANT NUMBER(s) DAAG29-78-G-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Systems Laboratory Department of Electrical Engineering Stanford University, Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE January 1982
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The research reviewed in this study specifically covered 3 issues:

1. The design of "ideal execution architectures" suitable for a range of source languages and host machine environments.
2. Analysis of the relation between source language program constructs and execution architectures.
3. Methods of implementing these "ideal" instruction sets.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DIRECTLY EXECUTED LANGUAGES

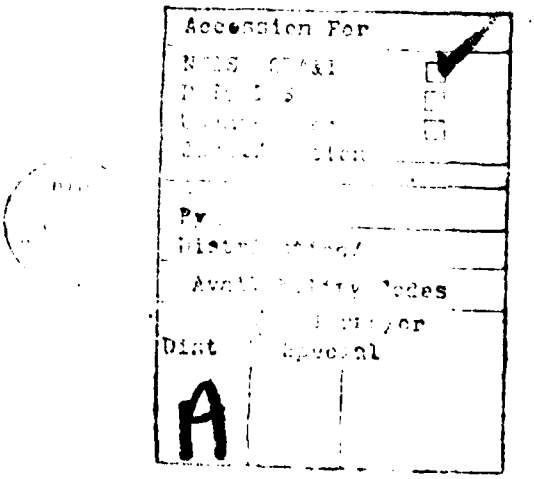
Abstract

The research reviewed in this study specifically covered 3 issues:

1. The design of "ideal execution architectures" suitable for a range of source languages and host machine environments.

2. Analysis of the relation between source language program constructs and execution architectures.

3. Methods of implementing these "ideal" instruction sets.



Introduction

The basic objective of this research is to find improved computer architectures which will allow more concise representations of programs, faster execution of programs and more accurate program representations.

In our past research, we have demonstrated techniques to afford significant improvement in spatial requirements to represent programs and in execution time to interpret programs. These techniques are based upon creating computer architectures which are tailored to a single high level language in which the architecture represents only those objects explicitly mentioned in the high level language (a 1:1 language to architecture mapping).

Techniques have been developed, such as accessing values by contour map and use of transformationally complete sets of formats, which allow the 1:1 mapping. The minimization of objects reduces both program size and improves execution time.

DEL Theory

We have completed at least a preliminary version of a broad based theory on the synthesis of directly executed languages. The kernel of this is a measure of ideal program representation. In actuality, this is a constructive measure of program space and interpretation that one can use for exploitation of alternative representations. This measure, while not necessarily achievable, intuitively represents a clearly superior program representation that is readily definable and easy to use.

We call these measures the *canonic interpretive measures* (*CI measures*) of a program. The space measure is the number of bits of program size, and the time measure is the number of instruction, data, memory and other actions in the program space.

The CI measure usually indicates a full order of magnitude less space than the System 360 representation or other familiar traditional machine representations. The CI measures play an important role in our development of a theory of DEL synthesis, since we essentially proceed in a step-by-step fashion to achieve program measures which are as close as possible to them. Except for frequency information, the CI measures are oriented toward an information theoretic minimum program representation. In many ways our DEL synthesis theory achieves exactly the CI measures. However, in order to achieve the transformational completeness property, we have found that 22 formats are required. This adds effectively about 5 bits to each instruction unit or statement ($\log_2$ of 22). In most other ways, however, CI measures are achieved.

Ideal Image Machines and their Hosts

In trying to define an ideal image machine, one must first recognize that the image machine architect is limited by the high-level languages that originally represent the program. If these are poorly formed, inefficient, or erroneous, there is no way that the architecture, the interpreter, or even the compiler can restore the initial deficiencies. The best the architect can do is to retain the information present in the high-level language representation and to provide this information in a form as concise and useful as the interpreter and the host require. Moreover, this interpretive hierarchy occurs at many levels in the system.

Thus, here we will derive certain aspects of HLL program representation which ultimately determine (and limit) the space-time product of image program interpretations. While we will come up with a number of measures of source program behavior, we must remember that these are measures and are not equally weighted. Their importance depends a great deal on the type of host machine doing the interpretation. These quantitative measures are expressed in architectural terms, so that for a particular machine representation, one could specify an ideal size and interpretation time for a source program, for example, and compare those ideal measures to the achieved size and interpretation time. Of course, specifying an ideal machine measure is a formidable task in itself.

Before proceeding we need to state some assumptions:

- Measures are independent of technology. We are interested in comparing logical representations and architectures, not in comparing different machine technologies.
- The original HLL source program is a good representation of the original problem; i.e., optimization is source to source. The original program is aready optimized to the degree desired.
- Measures focus on two aspects of program representation: space to represent the image program and time to interpret that representation. Simplicity in generating the representation is an implied necessity.
- The measures consist of correspondence, size, activity, stability and distance. The first two determine static program size. The last three affect in varying degrees the time it takes to interpret a program.

Canonic Measures of Interpretation

1. Correspondence. An ideal representation minimizes the number of objects to be interpreted without disturbing transparency with respect to the HLL source. For each semantic action in the HLL source (addition, subtraction, etc.), there is one instruction in the ideal representation (one *CI instruction*); for each *unique* name mentioned in the HLL statement, there is an explicit object identification in the ideal representation.

2. Size. All objects of the same class can be coded with variable-width identifiers appropriate to the language environment as follows:

- All variables are identified by $\lceil(\log_2$ of the number of variables in the environment)$\rceil$ bits,
- operations are identified by $\lceil(\log_2$ of the number of distinct actions specified in the environment)$\rceil$ bits, and
- labels are identified by $\lceil(\log_2$ of the number of distinct labels specified in the environment)$\rceil$ bits, where the environment is usually taken to be a single subroutine or procedure.

Ideally, the size measure requires that coding be as concise as possible, i.e., using $\lceil(\log_2$ of the number of objects in the environment)$\rceil$ bits. There are two aspects to coding. First, the objects should be of like kind. Thus, labels, operators, and values which can be easily distinguished should be treated and coded separately. Second, coding always takes place in the context of an environment. The statement itself can be an environment, although measures of time (such as measure 4, below) which count the number of environments interpreted will increase if such a small coding environment is chosen. This is a trade-off between two measures—space and time. The scope of the definitions used in the HLL usually defines the environment—that is, the environment encompasses names or objects of the same scope of definition, e.g., subroutines as follows:

3. Activity. This measures the number of objects interpreted. Usually, there are two separate types of activity—instructions (operations) interpreted and variables accessed from image storage:

- Let A_i be the number of instructions interpreted. Then, ideally, A_i is equal to the number of semantic actions specified (dynamically) in the HLL source.
- Let A_d be the number of data references required. Ideally, A_d is no more than the dynamic number of variables encountered.

4. Stability. This measures the number of environments encountered and other disruptions to the ordinary "in-line" processing of objects.

- Let S_e be the total number of environments encountered.
- Let S_c be the number of computed (interpreted) objects, e.g., array elements.
- Let S_b be the number of control actions (branches dynamically interpreted).

This measure is an extension of the activity measure in that it measures the more global types of activity, the number of environments encountered, the number of objects whose location has to be computed, and, finally, the number of branches or procedural statements encountered. Ideally, there is one interpretable action per HLL action.

5. Distance. This measures the "initialization" required in a program. The ultimate host machine might have an infinite cache, robust support of array accessing, and an unlimited branch target capture table. But even such a machine would be limited by the following measures:

- Let D_e be the number of unique environments entered.
- Let D_c be the number of unique objects requiring interpreted definition.
- Let D_b be the number of unique branch targets.

The distance measure represents the total number of unique environments and unique branches encountered. It represents the distance the program has transversed, i.e., the number of localities it has visited.

An Example and a Comparison

Consider the Pascal example shown in Table 1; *hardshuffle*—a program for shuffling vector elements the hard way—consists of shuffle procedure *swapvec* and the main program.

Swapvec interchanges the elements of two vectors from the first element up to the parameter *limit*. *Hardshuffle*—the main program—creates two vectors, *identity* consisting of the integers and *sum* which consists of the sum of the integers. For a variable *limit* ranging from *1* to *10*, *swapvec* is called to interchange the elements of the two created vectors. Finally, the values of the vectors are written out. Table 1 shows the derivation of the CI measures from the original *hardshuffle* source. The number of instructions, static (column a) and dynamic (j) correspond to the number of source semantic actions held in memory and encountered during execution. The column labelled "Data References" (k) is the dynamic count of activity (both local and main memory) for data objects. The number of syllables interpreted (o) corresponds to the count of the number of objects (variables, operations and labels) encountered in each instruction times its dynamic weight. The number of branches encountered (l) is a dynamic count of branch instructions, S_b, while the distance D_b measures the number of unique branch targets encountered. Computed data items (g) refers in this example to array elements. The dynamic sum of (g) is S_c (p) while the number of unique occurrences of such items is D_c (q). Of course there are two environments in this little example (D_e) and since the main program calls the subroutine ten times we have a total of 11 dynamic environments (S_e).

In computing the static CI measures of this program we first must identify the distinct operands and operators in each of the environments. For *swapvec* the operands are *a1*, *a2*, *limit*, *index*, *temp*, a container for the final value, and the constant *1*, i.e. 7 unique operands. The operators are **for**, array subscript, **end for**, and **return**—4 unique operators. There are also two labels. Each line of the body of *swapvec* basically represents a single CI instruction; an exception is

a1[*index*] := *a2*[*index*].

A separate instruction is required to compute *a2*[*index*] and retrieve its value. This value will be used in the next CI instruction which actually stores the value into *a1*[*index*]. In computing the number of operands per CI instruction it is merely necessary to follow the semantics of the statement, thus:

for *index* := *1* **to** *limit* **do**

has an index operand, two operands for the range (*1* and *limit*) and an operand for the final value. This particular statement also includes an opcode and a label (for use if *limit* < 1). Most other statements in this routine include a single container for each operand mentioned in the statement. Static program size for *swapvec* can be computed to be three bits per operand × 16 operands + 2 bits per operator × 7 operators + 1 bit per label × 2 labels, totaling 64 bits.

Similarly with the main program the unique operands are *i*, the operand for the final value, *identity*, *sum*, *swapvec*, and the constants *1*, *2* and *10*, for a total of 8 unique operands. The unique operators are array subscript, **for**, −, +, **end for**, *call*, *write*, and *writeln*, for a total of 8 unique operators. There are also 6 labels in the main program. Static size can be computed as 3 bits per operand × 45 operands + 3 bits per operator × 20 operators + 3 bits per label × 6 labels, the total being 213 bits.

The *hardshuffle* example illustrates the rather mechanical nature of deriving the CI parameters. First, the program is assumed to be optimized—although *hardshuffle* certainly isn't . Thus, the CI measures can be derived on a line by line basis. Each HLL statement will take at least one CI instruction. If the statement involves a computed data item (an indexed array element) an extra instruction sometimes is required. However, most HLL statements in the main program have one CI instruction in their execution architecture. Exceptions include a statement with two subscripted variables (line 4), the *write* instructions (lines 7 and 18) that use subscripted variables, and the statement computing *sum*[*i*] (line 13) which requires 4 instructions (−, + and subscripting). In computing operands, usually the count is simply the number of explicit variables mentioned in the statement. E.g. on line 2 we have: *index*, *i* and *limit*. Occasionally the semantics of an operation requires a hidden operand; e.g. (line 6) the **end** of a **for** requires a final value operand. The statement on line 3 has but one instruction while line 4 required two instructions. In column (a) we enumerate the number of instructions per statement, and column (b) enumerates the number of associated operands per statement. Labels are not tabulated explicitly but the sum of column (a) plus column (b) plus implied labels equals the number of syllables, column (c), for that line. Thus on line 2, we see a single instruction with three associated operands (*index*, *i*, and *limit*). There is an implied label; thus there are five syllables to be interpreted for instruction execution. On line 4 it is assumed that one instruction computes the location of the indicated source operand and fetches it. The second instruction computes the location of the sink operand and stores the value of the source operand. Thus the two instructions

Line		← counts per statement →										← CIM →							
		instructions (a)	operand (b)	Memory Reads (c)	Memory Writes (d)	Syllables (e)	Branches (f)	Computed Data items (g)	Environment (h)	Dynamic Weight (i) (No. of times executed)	Instructions executed (j)							Computed data: S_c (p)	First occurrance: D_c (q)
											axi	bxi	fxi	cxi	dxi	exi	gxi		
											Σ=435	Σ=830	Σ=108	Σ=139	Σ=130	Σ=1298	Σ=269		
1	Begin	0	0	0	0	0	0	0	1	10	0	0	0	0	0	0	0	0	
2	For index := 1 To limit Do Begin	1	3	0	0	5	1	0	0	10	10	30	10	0	0	50	0		
3	temp := a1[index];	1	3	1	0	4	0	1	0	55	55	165	0	55	0	220	55	55	
4	a1[index] := a2[index];	2	4	1	1	6	0	2	0	55	110	220	0	55	55	330	110	55	
5	a2[index] := temp	1	3	0	1	4	0	1	0	55	55	165	0	0	55	220	55	0	
6	End (*For*)	1	1	0	0	3	1	0	0	55	55	55	55	0	0	165	0	0	
7	End (*swapvec*);	1	0	0	0	2	1	0	0	10	10	1	10	0	0	20	0	0	
8	Begin (*hardshuffle*)	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	
9	identity[1] := 1;	1	3	0	1	4	0	1	0	1	1	3	0	0	1	4	1	1	
10	sum[1] := 1;	1	3	0	1	4	0	1	0	1	1	3	0	0	1	4	1	1	
11	For i := 2 To 10 Do Begin	1	3	0	0	5	1	0	0	1	1	3	1	0	0	5			
12	identity[i] := i;	1	3	0	1	4	0	1	0	9	9	27	0	0	9	36	9	9	
13	sum[i] := sum[i - 1] + i	4	6	1	1	10	0	2	0	9	36	54	0	9	9	90	18	9	
14	End;	1	1	0	0	3	1	0	0	9	9	9	9	0	0	27	0	0	
15	For i := 1 To 10 Do swapvec(identity, sum, i);	3*	7	0	0	5+4 +3	2	0	0	1+10 +10	21	3+3 +10	1+0 +10	0	0	5+40 +30	0	0	
16	For i := 1 To 10 Do Begin	1	3	0	0	5	1	0	0	10	1	3	1	0	0	5	0	0	
17	write(identity[i]);	2	2	1	0	4	0	1	0	10	20	20	0	10	0	40	10	10	
18	write(sum[i]);	2	2	1	0	4	0	1	0	10	20	20	0	10	0	40	10	10	
19	writeln	1	0	0	0	1	0	0	0	10	10	0	0	0	0	10	0	0	
20	End (*For*)	1	1	0	0	3	1	0	0	1	10	10	10	0	0	30	0	0	
21	End (*hardshuffle*)	1	0	0	0	2	1	0	0	0	1	0	1	0	0	2	0	0	

```
Type indextype = 1..10;
     vector = Array [indextype] of integer;
Var identity, sum: vector;
    i: indextype;

Procedure swapvec(Var a1, a2: vector; Var limit: indextype);
   Var index  indextype;
       temp; integer;
```

*one instr. for FOR; one for swapvec call and one for implied RETURN

Table 1: Deriving the CIF for Hardshuffle

	Pascal				System 370 Pascal		System 370 PL-1	
	CIM	Adept	PDP-11	P-Code	without linkage	with linkage	without linkage	with linkage
Static Size (in bits)	277	1184	2800	4960	3056	4288	3668	4952
Static Number of Instr.	27	27	105	155	90	99	130	168
Dynamic Number of Instr.	435	435	2023	2404	1481	1536	2322	2522
Total no. of data references (local + global/computed)	830							
No. of main memory (read) refr.	139	139	963	2797	443	486	1450	1626
No. of main memory (write) refr.	130	130	479	2179	402	402	457	503
No. of Syllables interpreted	1298	1923	10,115	9976	7393	7668	11,318	13,198
No. of Branches dynamic: S_b	108	108	230	202	219	261	188	211
No. of Branches first target encounter D_b	14	14	15	19	18	24	23	28
No. of computed data refr.: dynamic S_c	269	269	-*	269	140	140	556	556
No. of computed data refr.: unique encounters D_c	149	159	-*	269	140	140	556	556
No. of environments: dynamic S_e	11	11	41	11	41	41	81	81
No. of environments: unique D_e	2	2	5	5	2	5	8	8

*not calculated--see text.

Table 2: Comparison for Various Architectures With CIM

consist of a total of four operand fields, giving a total of six syllables to be interpreted for their joint execution. On line 6 the end of the for loop requires one instruction—a return and an implied operand which contains the final value as well as a label associated with the return. Thus the counts for the end of the procedure depend upon the sematics of the instruction sequence initiated. On line 7, for example, the *end* requires no operands; hence only two syllables are interpreted. The resulting columns (a) through (h) are weighted by column (i), the dynamic count or number of times each line is executed. When the respective columns are weighted we can derive the appropriate CI measures for that line. Summing over all lines we have the total CI measures for the program.

Main memory read and write references—columns (c) and (d)—arise from computed data references. The principle difference between operand references (b) and main memory references—column (c) plus column (d)—is that the operands are assumed to access a local memory if they are local scalars—other references go to main (or global) memory. The counts in columns (c) and (d) are included in column (k)—data referencing; these give an indication of the predictability of referencing activity. An architecture which does not accommodate local references is more likely to encounter memory bandwidth and contention problems. The total number of data references—column (k)—is simply the number of operands per instruction weighted by the number of times that the instruction is executed. The sum of column (k) for the program is 830 references. Register-oriented architectures as well as architectures which make provision for rapid access to local variables should do better than this, but reducing references below the main memory referencing activity—column (m) plus column (n)—should be more difficult.

Since each instruction called for in column (a) requires an op-code syllable and each operand (d) requires a syllable, the total number of syllables per statement (e) is the sum of (a) and (b) and the number of labels. Labels are not tabulated directly. The branches (f) and computed data item counts (g) are fairly obvious. Line 15 represents an initial (for) branch, a call to *swapvec* and a return.

Table 2 is a complete evaluation for the example *hardshuffle* for all of the CI measures on a variety of different architectural approaches. A fair comparison for a variety of architectures is a more formidable task than might first appear. The measures are significantly influenced by compiler strategies and run time environments as well as the basic architecture itself. Thus, the data in Table 2 requires some explanation.

The first comparison is with an execution architecture called Adept, developed at Stanford and derived from principles of minimizing CI measures while maintaining transparency for Pascal programs. By using an additional format syllable in each instruction it matches most static and dynamic CI instruction count measures. It also matches the CI measures for memory activity as well as most of the stability and distance measures. Additional syllables per instruction add about 5 bits to each instruction and hence account for

about 110 additional bits in static program size. An additional 800 bits of Adept are used to hold addresses for subscripted variables, array bases, and other environmental data. An Adept variable reference consists of the addition of an environmental pointer to a variable index whose container matches the $\log_2$ CI requirement. Each environment then will have its own environmental pointer and container width. Some variables such as array elements have an address computation before the element can be retrieved from main storage which contains the image array. Thus the address of the base of the vector must be stored as well as the retrieved array element. The additional Adept space includes these address constants and other values containing information for the routine to execute properly. The object code for Adept is based upon a 1 + 1 pass compiler developed by S. Wakefield. The Adept system has a simple run time environment required for a small stand alone system.

The pdp-11 figures are based upon the Pascal compiler developed at Vrije University (the Netherlands), Pascal-VU. It produces an intermediate program representation EM-1, developed by A. Tannenbaum, which it further translates into pdp-11 code.

The static program size is the size of the instruction stream; however, in the pdp-11 architecture many of the data parameters are represented as immediate data in the instruction stream. The number of computed data references was not tabulated for the pdp-11 because of the nature of the architecture—*the pdp-11* architecture is heavily oriented towards conditional interpretation of successor syllables. Overlapping of instructions and/or syllable interpretation would be an extremely difficult process at best; therefore the total amount of sequentiality in the instruction stream is not a measure of the architecture's responsiveness to a particular example. The additional number of environments encountered in the pdp-11 object code is a direct result of the *write* commands in the subroutine. These are implemented in the object code as calls to the operating system.

Before discussing the P-code or 370 architectures the computation of the computed data references should be examined. There is no problem in counting the occurrence of array elements in the source code or in fact in a close surrogate to source code such as the Adept code. In more traditional machines the occurrence of a computed data reference is a more murky event. It manifests itself as the loading of an index register or an indexed value in one instruction, and the use in the immediately following instruction of that value as a parameter in an address computation. In a stack machine such as the P-code processor a value is placed on a stack which is used in the next instruction as an address. In System 370 the following is a typical instance:

```
LR 15, A(13,14)      instruction i
LR  2, B(15,10)      instruction i+1
```

In the above, a value is loaded into a register (15 in this case) and used as an index/base value in the

immediately following instruction. The result of this event is a potential "break" in the pipeline. The address of the operand in instruction i+1 cannot be computed until instruction i is fully executed. This delay is reflected as an increased execution time for instruction i+1 in overlapped machines, especially where this deferred data access is in the vicinity of a conditional branch.

In computing the number of environments, we count the number of different environment change indicators, such as the Branch and Link (BAL) instruction (System 370). The called routine while outside the object code listing may call other environments and are not accounted for in our counts, a fact that we shall discuss shortly.

The P-code machine is actually a surrogate for the Pascal language. It is a stack oriented machine and meant to be a transportable media for Pascal programs. Any host machine can compile into P-code from a Pascal source and (in theory at least) another machine equipped with a P-code interpreter can execute this compiled code. The emphasis for most P-code compilers is rapid compilation; thus the P-code statistics are derived from a non-optimized compiler—much the same in philosophy as Adept.

The large number of dynamic instruction occurrences for P-code when compared to Adept - over 5 to 1—is largely accounted for by the *push* and *pop* instructions inherent in a stack machine. Notice that the dynamic number of P-code branches, for example, is less than a factor of two to one over the CI measure.

In comparing the System 370 to any of the other environments one is faced with immense problems. So far we have been discussing machines and measures in very limited run-time environments with relatively minimal generality in support for non-Pascal system facilities, the antithesis of the generalized support provided by the 370 Operating System. While the 370 program size itself is 3056 bits this excludes prolog, epilog and data space which alone—through a standard interface—is reserved at 16000 bits. This overwhelms our comparison and since it contains or allows for a great deal more information handling and communications than required either in this program or by any of the other architectures, we largely eliminate (insofar as possible) instructions or data areas which are not specifically associated with the program *hardshuffle*. The column labelled "without linkage" represents the additional number of instructions in the minimum linkage path between the two routines. Excluded from this are the instructions executed as part of the linkage which are calls to common run time facilities, space allocation, etc. These are again excluded in our comparison since it seemed to us that the inclusion of such data is more a measure of run time philosophy and its generality than a measure of architecture itself. Calls to such facilities during routine entry are not counted in the environment counts either. To fully include all instructions executed in a typical System 370 program plus all data areas and prolog and epilog areas would increase the cited numbers by several times. Thus, the 370 numbers can be interpreted as minimum numbers in comparing with the other architectural

figures. The 370 numbers reflect an estimate of the measures of the architecture in a very simple dedicated runtime environment which simply is not available to us to measure. As a further experiment on 370 the *hardshuffle* source program was recoded in PL/I and recompiled using an optimizing PL/I compiler. The increased generality of PL/I plays a role in limiting the compiler's ability to optimize the program. The additional environments introduced in the PL/I version of the program result from the compiler causing several environment changes per source *write* command.

It is interesting to note that at least for this example the more dramatic variations in architectural measures occur in parameters—such as space, dynamic instruction count, and syllables interpreted—that affect simpler hosts, particularly partially mapped and well mapped machines. Parameters such as stability and distance remain relatively invariant from the canonic measures over the spectrum of architectures considered. In fact, compilers seem to play a more significant role than the architectural arrangements themselves. This supports the observation that is more or less a truism that compiler technology is even more important than the architecture as the interpreter and executor technology is enhanced, while for simpler interpreters (hosts) the architecture seems to play a dominate role in determining execution performance.

Scientific Personnel Supported by this Contract

Michael J. Flynn, Principal Investigator
Professor, Electrical Engineering
Stanford University

Jerome Huck
Research Assistant
Stanford University
(Expected to receive Ph.D. degree in Electrical Engineering, Summer Quarter 1982)

Ruby Lee
Research Assistant
Stanford University
(Received Ph.D. degree in Electrical Engineering, Summer Quarter 1980)

Charles Neuhauser
Research Assistant
Stanford University
(Received Ph.D. degree in Electrical Engineering, Spring Quarter 1980)

Scott Wakefield
Research Assistant
Stanford University
(Expected to receive Ph.D. degree in Electrical Engineering, Summer Quarter 1982)

Robert Wedig
Research Assistant
Stanford University
(Expected to receive Ph.D. degree in Electrical Engineering, Summer Quarter 1982)

Clark Wilcox
Research Assistant
Stanford University
(Received Ph.D. degree in Computer Science, Summer Quarter 1980)

Banman Zargham
Research Assistant
Stanford University

Technical Reports and Publications Sponsored under this Contract

"A Theory of Interpretive Architectures: Ideal Language Machines" by Michael J. Flynn and Lee W. Hoevel, Computer Systems Laboratory TR No. 170, Stanford University, Stanford, CA, February 1979.

"A Theory of Interpretive Architectures: Some Notes on DEL Design and a FORTRAN Case Study", by L. W. Hoevel and M. J. Flynn, Computer Systems Laboratory TR No. 171, Stanford University, Stanford, CA, February 1979.

"Instruction Stream Monitoring of the PDP-11", by Charles J. Neuhauser, Computer Systems Laboratory TN No. 156, Stanford University, Stanford, CA, May 1979.

"EMMYXL—An Assembler for the Stanford EMMY", by Bahman Zargham, Computer Systems Laboratory TN No. 164, Stanford University, Stanford, CA, September 1979.

"Some Notes on a DEL Basis for Language-Oriented Operating Systems", by Michael J. Flynn and Martin Freeman, Computer Systems Laboratory TN No. 169, Stanford University, Stanford, CA, November 1979.

"Analysis of the PDP-11 Instruction Stream", by Charles J. Neuhauser, Computer Systems Laboratoy TR No. 183, Stanford University, Stanford, CA, February 1980.

"L ections and Issues in Architecture and Language", by Michael J. Flynn, Computer Magazine, Vol. 13, No. 10, October 1980, pp. 5—22.

"Dynamic Detection of Concurrency in DO-loops Using Ordering Matrices", by Robert G. Wedig, Compute Systems Laboratory TR No. 209, Stanford University, Stanford, CA, May 1981.

"Execution Architecture: The DELtran Experiment", by Michael J. Flynn and Lee W. Hoevel, Accepted for publication in IEEE Transactions on Computers.

"Ideal Execution Architecture", by Michael J. Flynn and Lee W. Hoevel, Submitted for publication in IEEE Transactions on Software Engineering.

"Analysis of Architectures for High Level Languages", by M. J. Flynn and J. C. Huck, Submitted to and presented at the HLL Computer Architecture Workshop, October 1981, Los Angeles, CA.


ILMED
8